Science of Computer Programming 74 (2009) 399–413

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Combining preprocessor slicing with C/C++ language slicing☆

László Vidács*, Árpád Beszédes, Tibor Gyimóthy

University of Szeged, Department of Software Engineering, Árpád tér 2., H-6720 Szeged, Hungary

ARTICLE INFO

Article history:
Received 1 September 2008
Received in revised form 10 February 2009
Accepted 18 February 2009
Available online 3 March 2009

Keywords:
Program slicing
C/C+
Preprocessing
Preprocessor slicing

ABSTRACT

Of the very few practical implementations of program slicing algorithms, the majority deal with C/C++ programs. Yet, preprocessor-related issues have been marginally addressed by these slicers, despite the fact that ignoring (or only partially handling) these constructs may lead to serious inaccuracies in the slicing results and hence in the program analysis task being performed. Recently, an accurate slicing method for preprocessor-related constructs has been proposed, which – when combined with existing C/C++ language slicers – can provide more complete slices and hence a more successful analysis of programs written in one of these languages. In this paper, we present our approach which combines the two slicing methods and, via practical experiments, describe its benefits in terms of the completeness of the resulting slices.

1. Introduction

Different program analyses and analysis tools have been proposed to assist activities related to software maintenance. Program slicing in particular [1–3], is an analysis procedure where parts of a software system are extracted which represent a sub-computation of interest, thus reducing the size and complexity of the problem being addressed. A short introduction to program slicing and its relation to our present work is given in Section 2.

There are a number of challenges which make the creation of practically usable program slicing tools difficult. Some of these are very general and have kept the research community busy for decades, while there are also platform specific issues that are specific for one programming language or a family of languages or platforms. This paper deals with one particular issue, namely the *preprocessor*, in the context of computing program slices for C/C++ programs. In many different program analysis fields researchers cite the preprocessor as an obstacle to implementing correct analyses, *e.g.,* [4,5]. Unfortunately, the situation is no better with program slicing.

Alas, preprocessor issues are often completely neglected by slicing algorithms, or at least, handled rather poorly. Features like file inclusion or conditional compilation are sometimes handled in an acceptable way, but macro expansion, for instance, is a different story. The best that existing slicers can do is to mark those program points originating from macros and display this information on the screen. CodeSurfer [6], for instance – which is probably the best-known static C/C++ slicer available today – displays information on macros appearing in slices, but is unable to include them in the slicing process itself. However, ignoring the existence of dependencies between preprocessor constructs and language elements may lead to serious errors in certain tasks where program slicing is applied. For example, in an incremental software development scenario, a change to a macro definition should be propagated throughout the system which will, in many cases, involve other macros and regular language elements as well. Impact analysis using slices that do not include preprocessor elements will be inaccurate and so potentially unsuccessful in situations like these.

☆ A preliminary version of this paper has been presented at the 16th IEEE International Conference on Program Comprehension in Amsterdam, The Netherlands, June 10-13, 2008.

* Corresponding author.
E-mail addresses: lac@inf.u-szeged.hu (L. Vidács), beszedes@inf.u-szeged.hu (Á. Beszédes), gyimi@inf.u-szeged.hu (T. Gyimóthy).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.02.003

In this paper, we will present a possible way to implement such a preprocessor-aware C/C++ slicer. It is based on the so-called *macro slicing* method introduced earlier [7,8]. Essentially, a macro slice is a set of dependencies between macro definitions and their uses, which is fairly similar to other notions of dependency-based slices. These macro slices are then combined with traditional language slices thus providing a more complete dependency set for a specific slicing task. Our slicer is experimental and is based on existing tools, applying specifically designed algorithms for producing the combined slices. The article is an extended version of a conference paper [8] with the following main contributions:

- More details on the algorithms and the implementation of the connection itself,
- possible applications and motivating examples are given, and
- the experiments have been repeated more thoroughly with a bigger set of programs.

In Section 3, we will justify the need for preprocessor-aware C/C++ slicers by providing motivating examples and application scenarios. The problems with existing implementations and other related work are listed in Section 4, then we describe our approach in detail in Section 5. Section 6 deals with our current implementation, then we present experimental results in Section 7. In the last section we will draw some conclusions and make some suggestions for future study.

2. Program slicing

Program slicing is an analysis method for extracting parts of a program which represent a specific sub-computation of interest. It has been originally introduced by Weiser [2] to assist debugging, where a set of program points is sought for, which affect the variables of interest at a chosen program point, called the *slicing criterion*. The reduced program is called a *slice*. This definition is sometimes more precisely referred to as *backward slice*, since – having procedural programs in mind – it associates a slicing criterion with a set of program locations whose earlier execution affected the value computed at the criterion. On the other hand, a *forward slice* is a set of program locations whose later execution depends on the values computed at the slicing criterion. Slicing can also be categorized as *static* or *dynamic*. In static slicing, the input of the program is unknown and the slice must therefore preserve meaning for all possible inputs. By contrast, in dynamic slicing, the input of the program is known, and so the slice needs only to preserve meaning for the input under consideration.

Over the years, a number of algorithms to compute program slices has been developed; for an overview see [1,3]. One of the most cited approaches is to apply a pre-computation step in which a representation of the program under investigation is constructed first, which captures the *dependencies* among program elements (for instance, data dependencies). This representation is called the Program (or System) Dependence Graph, whose basic form for static slicing and procedural languages was given by Horwitz *et al.* [9]. The nodes of this graph represent the program elements (instructions), while the edges connecting them correspond to the program dependencies. The counterpart of this graph for dynamic slicing, the Dynamic Dependence Graph [10] includes a distinct vertex for each occurrence of a statement in the execution of the program on the input under consideration (called the execution history). Eventually, the computation of a slice with these approaches means finding all reachable program elements in these graphs starting from the slicing criterion.

In this work we reuse the basic slicing principles to compute *macro slices* by constructing the *Macro Dependency Graph (MDG)* first [7]. But, some slicing concepts need to be reinterpreted in the scope of macro slicing, as discussed in the following. In their first approach, Agrawal and Horgan introduced dynamic slicing by refining the static Program Dependence Graph using information from the execution history [10]. The need for the Dynamic Dependence Graph to construct accurate dynamic slices was then demonstrated by the authors. Namely, a distinct node for each occurrence of an instruction was implied by the loops in execution history. In the case of macro slicing the set of *mcall* edges (discussed in Section 5) serves as execution history. The history of macro invocations can be reconstructed based on them (if a macro body contains more than one macro invocation, their order in history is the order of appearance in the macro body). Fortunately, there are no cycles in macro calls, so it is not necessary to create new macro definition nodes for each call.

Similar to other forms of slicing, we use the notions of *forward* and *backward* for macro slices as well. We should mention, however, that our choice for this terminology was rather arbitrary. In the case of procedural programs the slice direction is defined with respect to the *order of computations* in the program. But in the case of macro programs, the notion of "order" is less obvious as there are no "executable instructions" (consider, for example, the fact that the macro dependency edge points in the opposite direction to the macro call edge, while with procedural programs the control flow aligns with the control dependency). Furthermore, it is also meaningless to talk about data dependencies for macro slicing, since these may exist only between the actual arguments and the formal parameters, but the macro definition itself is not a part of the program, and hence the data dependency starts from the point of the initial call and necessarily ends at the same place.

3. Motivation, utilization

3.1. Motivating example

Macro slices can be used, among other things, for the purpose of change impact analysis. The developer usually has to carry out small changes during the system maintenance tasks, but in a large software system the effect even of a small change is hard to predict. Let us assume that the small program part to be altered is a macro definition. Our first motivating problem is to find the points of a C/C++ program which are affected by a modified macro definition. The modified definition

```
1: #define ASSIGN(v) = v
2: #define SGN unsigned
3: #define DECLI(name, val)  SGN int name ASSIGN(val);
4: DECLI(i,2)
5: printf("%u\n",i);
```

Fig. 1. Small example on macro and regular forward slices.

```
1:
2:
3:
4: unsigned int i = 2;
5: printf("%u\n",i);
```

Fig. 2. The example source code after preprocessing.

may be used in (called from) other macro definitions, which can be called again from many points of the program. Next, the calls that use the definition are replaced and become part of the C/C++ language constructs. But these constructs may affect other parts of the program, which may be captured by traditional C/C++ language slices. In other words, the affected part of a program consists of *both* preprocessor-related elements and C/C++ program elements. The union of the forward macro slice starting from the given definition and the forward language slice starting from replaced parts gives all the affected points. A small example which illustrates this is given in Fig. 1.

The slicing criterion for macro slicing is the macro definition in line 1. The corresponding macro slice contains lines 1, 3 and 4, while the macro call in line 4 is the link between the two kinds of slices. During preprocessing, the macro call `DECLI(i,2)` is expanded to `unsigned int i = 2;`, which is a C/C++ program element. The replaced macro is the slicing criterion for C/C++ language slicing, and the language slice contains lines 4 and 5. The combined slice contains all lines of the example code except line 2, which means that changing the macro definition on line 1 affects four lines. A failure to identify these additional dependencies may cause a problem in change impact analysis, for instance.

The procedure of combining slices works in the other direction as well. Fig. 2 lists the previously shown example code after the preprocessing phase. The macro definitions are hidden from the compiler. Let the slicing criterion contain the variable `i` in line 5. The C/C++ backward slice algorithm does not know about macros as the slice contains lines 4 and 5 only. Using the fact that line 4 comes from macro replacement, a backward macro slice can be computed on line 4, which contains lines 4, 3, 2, 1. The combined backward slice contains every line of the original example, instead of two lines of the C/C++ slice. An example where this can cause problems is when this additional information is not available in a debugger, the user could not track down to all possible causes of an error which is being debugged.

3.2. Real world example

How useful the combined slices may be is illustrated by the following example taken from the *flex* subject program of our experiments section. Let us assume that a new functionality is added to our software system and that we have to modify (among other things) the part of the program related to memory handling. It turns out that some part of the code to be modified contains a macro call in the original source. Using the macro backward slicing, the used macro definitions can be accurately located.

Let us assume that modifications are done on the macro definition `reallocate_integer_array()` found at `flexdef.h` line 686:

```
#define reallocate_integer_array(array,size) \
  (int *)reallocate_array((void *)array, size, sizeof(int))
```

Note that the "called" `reallocate_array()` is not a macro, but a C function. The following task is to build the whole program and test. Two problems may arise. The altered module compiles, but why do we have building problems for a totally "unrelated" module? And having modified the macro definition, which parts of the program have to be tested?

In the case of modifying a C function, slicing can be used to determine dependent program parts, to give hints on affected files/modules, so one may select the appropriate test cases instead of performing full program test. In this case the combined forward slice on the changed definition would help. The macro slice shows that 31 toplevel macros are involved. The macro definition change is done based on one part of the program but there are 30 others where we have to test. An example path in the slice is when the definition is called from the `DO_REALLOCATION(dfa.c:261)` and `PUT_ON_STACK(dfa.c:269)` macros. The file `dfa.c` at line 308 contains a simple macro call, namely `PUT_ON_STACK(ns)`, but when the source is preprocessed, it is replaced by a do-while loop which is 358 characters long. One macro change goes through 31 points in the source, for each C slice must be computed, which finally shows that 8271 source lines may be affected. The 31 toplevel macros show where to check the correct macro usage, helping in build problems (sometimes in different modules). While the full combined slice gives hint on which part of the program is affected, which allows for using selective retesting to reduce maintenance costs.

3.3. Utilization

The basic property of the approach proposed here is the handling of macros. Generally speaking, the method is usable for the same purposes as C/C++ slicing: change impact analysis [11], program decomposition [12], software re-use [13,14], debugging [15] and regression testing [16,17]. Dependencies added by the macro slices provide more accurate analysis and hence better results. In the case of backward slices the C/C++ slice is continued with macros, bringing source files into the slice that had not previously been taken into account. The special case of backward slicing is presented in the above example, where the backward slice was taken for a macro call to identify the used macro definitions. Forward combined slices start at a macro definition, which cannot be located using pure C/C++ slicing.

The preprocessor-related program constructs deserve more attention from the utilization point of view. The backward direction can be used when the programmer encounters a macro call in the code, and neither the replaced value nor the used macro definitions are visible, which would help in debugging (the place of the compiler error is a macro call). This is true for program comprehension as well: the simple macro call is expanded to many C/C++ constructs like that shown in the previous example. As already shown, selecting the right test cases can be helped using our method as well.

The current implementation of the macro slicer works on just one configuration, which is analyzed by the C/C++ slicer. This keeps the result synchronized, but also means that in general the toolset is not suitable for solving configuration-related issues. However, conditional directives usually contain macro checks (using the *defined* operator), which are included into the macro analysis. Thus the forward slice requested on a macro definition which determines the configuration (e.g. `#define USE_SMART_PTR`) will provide a hint about which part of the current configuration is configuration dependent. Unfortunately the macro call in a conditional directive is not matched to any C/C++ language element (see Section 6). A new dependency between conditionals and C/C++ elements would help. The current implementation of the macro analyzer contains a dependency relation like this, but it has not yet been used in macro slicing.

The data structure employed for macro slicing (introduced in [18]) can be configuration dependent (as used in this work) or configuration independent. The latter has not yet been implemented, but in the future may open the door for configuration independent macro slicing. The C/C++ part seems to be the harder problem though as the C/C++ slicer produces slices for just one configuration, but the configuration independent combined slicer should run the C/C++ slicer for every possible configuration and connect/merge the results. Checking every possible configuration can be usually approximated by some important configurations, but right now a configuration independent slicer is just the subject of future research.

4. Related work

There are relatively few slicing tools available for C/C++ programs. Binkley and Harman [19] conducted an empirical study of static slice size of C programs and they mention three general purpose slicing tools: Unravel [20], Sprite [21] and CodeSurfer [6], using the latter in their experiments. Unravel was a research prototype that was developed in a discontinued project. It has a number of deficiencies including the fact that it can only accept preprocessed ANSI C code, which makes it clear that handling macros has not been implemented. Sprite implements some enhancements to traditional slicing algorithms, most notably in the field of points—to data. Since the tool is not publicly available and the related publications do not deal with this issue, it is not clear how macro dependencies are handled by using this approach.

The commercial slicing tool CodeSurfer, marketed by GrammaTech Inc., is probably the most up to date and best developed slicing program for C/C++ today. It is able to compute various static dependency data by employing the latest code analysis and program slicing technologies. However, it also has modest support for handling preprocessor-related artifacts. It is able to identify the location of macro definitions and uses and present this data to the user. However, it is not possible to compute slices using macro definitions as criteria. Furthermore, the slices will only include statements that exist after macro expansion. Nevertheless, we used this tool in our experiments because the information supplied by CodeSurfer about macro usage was sufficient to implement our approach.

The APP (Abstract PreProcessor) defines an abstract language to handle preprocessor directives in a similar way to other programming languages. Handling directives in a consistent way allows one to perform an analysis such as slicing as a solution for some preprocessor-related problems. The example presented on slicing is similar to our backward macro slicing, but it has the advantage of telling one the conditional directives in the path. Unfortunately, the implementation drawbacks prevent this tool from being applied to real C programs (e.g. the function-like macros are not supported).

The Ghinsu software maintenance environment is the most closely related tool to our approach [23]. With it, by clicking on a macro invocation the called definitions are highlighted (backward macro slice using our terms). In addition, it supports both static and dynamic slicing, ripple analysis and other program analyses on ANSI compliant C source code. This tool also utilizes a dependency graph in which the tokens of preprocessed code are classified according to whether (and if so, how) they are involved in macro expansion. Unfortunately, it appears that this project has been discontinued, and from the latest information gleaned we found that the implementation has certain drawbacks which prohibits its use from being applied to real programs. For example, certain language features and complex projects consisting of multiple source files are not handled.

There are also some other tools which are not slicers but have quite similar functionality aspects for the comprehension of macro usage. The GUPRO program understanding framework implements a macro folding mechanism where a macro

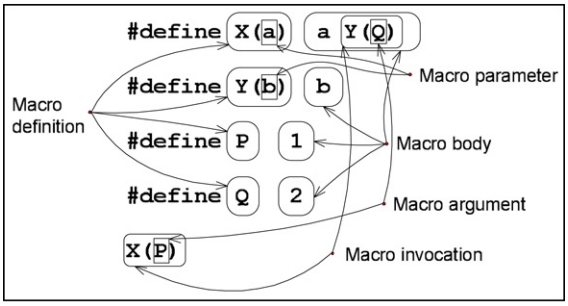

Fig. 3. Example macro call.

can be hidden or revealed at the place of the call [24]. The Understand for C++ reverse engineering tool provides cross-references between the use and definition of software entities [25]. This includes the step-by-step tracing of macro calls in both directions. The user can trace back the uses of a given macro definition, but the information obtained is not accurate in some situations. These tools however do not incorporate C/C++ language slicing as we do in our approach.

Finally, an interesting topic for future research is the investigation of the so-called dependence clusters [22] on preprocessor slices. A dependence cluster is a set of program statements all of which are mutually interdependent. Dependence clusters are approximated by the set of statements which have similar slice sizes. In the case of combined slicing the similar slice sizes are also observed. Macro slices are, however, usually short, hence the C/C++ slices are dominant in the combined slices. In the preprocessor case, macros with really short macro slices do not necessarily belong to the same cluster. This issue deserves more careful examination, however.

5. Combining C/C++ preprocessor and language slices

The process of combining the two types of slices can be performed in both the forward and backward directions. In the forward direction the slicing criterion is a macro definition. The macro slice contains toplevel macro calls as connection points, the replaced toplevel macro calls are (part of) C/C++ program elements, whose program elements serve as slicing criteria for regular language slicing. The final slice contains both preprocessor and C/C++ program elements. The backward direction is similar but here the slicing criterion is a C/C++ program element, and the language slice may contain program elements which are in turn parts of the result of a macro call. These macro calls are used for macro slicing and the final slice contains the language slice and all the macro slices as well.

In this section we first provide a brief account of macro slices and then describe our approach for combining the two kinds of slices.

5.1. Macro slices

Here, just an overview will be presented along with some figures and definitions. For a detailed description of macro slices we refer to our previous work [7].

Slices are usually defined on a graph structure which represents dependency relations between program elements. Accordingly, the structure of macros is defined by using sets and relations, and a dependency graph is defined based on macros using a dependency relation which is appropriate for slicing macros.

The terms used to formalize the macro replacements are included in the example given in Fig. 3 (the macro call results in `1 2`). Note that the captions here are just for illustrative purposes, and some arrows have been omitted from the picture.

- *macro definition* — the place of the `#define` directive. The definition consists of three parts, namely *macro name*, optionally *parameters*, and *macro body* (also called the replacement list).
- *macro invocation* — the place in the program where a macro name is used (where the name is to be replaced with the macro body from the definition).
- *macro expansion* — the process of a single macro replacement, where macro arguments are also expanded and replaced.
- *full macro expansion* — all macro expansions which are necessary to get the final result of an initial macro expansion (including the macros in the re-expansion process of macro bodies).
- *toplevel macro invocation* — starting point of a full macro expansion (a full macro expansion necessarily starts outside the `#define` directives).

Let us construct a set called *MC* which contains macro invocation nodes and macro definition nodes. Both types of nodes are multinodes (node sets) in the sense that they contain many preprocessor elements, but for the sake of simplicity and readability we shall treat them as one node. The first type is based on toplevel macro invocations (depicted by a black node in Fig. 4): each node contains a toplevel invocation and the invocations which are in its arguments. The second type is based on macro definitions: each node contains a macro definition and the macro invocations contained by its macro body. The

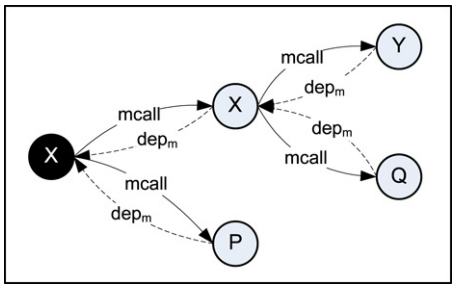

Fig. 4. The *mcall* and the dep_m relations on the simplified *MC* set.

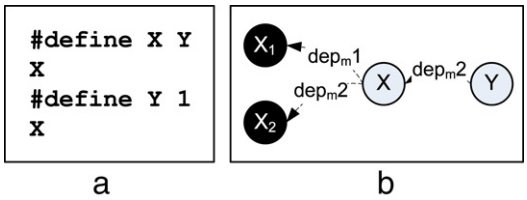

Fig. 5. Example code and *MDG*: (a) program code (b) *MDG* with edge coloring.

set is constructed in order to exclude every relation other than macro calls. Based on the macro calls, the macro dependency relation can be defined on the *MC* set as the inverse relation of the call in the following way ($y \in mcall(x)$, where $x, y \in MC$ means that there is a macro call in x that calls definition y):

Definition 1. Let $dep_m \subseteq MC \times MC$ be a relation, $b \in dep_m(a)$ if and only if $a \in mcall(b)$, where $a, b \in MC$.

An example set with relations is given in Fig. 4. The macro dependency relation points in the opposite direction to that of the *mcall* relation. Informally, a node is macro dependent on another if and only if there is a macro call from inside the first node to the second node.

Let us construct the Macro Dependency Graph (*MDG*). At this point we again refer to [7] where more information is given about macro dependencies. The nodes of the graph are the elements of the *MC* set and the directed edges are created from the dep_m relation. The edges are multiple edges because there may be more full macro expansions which have a common subset of dependency edges, but we have to distinguish them. Edge coloring is used to sign the edges that belong to a particular full macro expansion.

Definition 2. Let $MDG = (MC, E, I, C)$ stand for the Macro Dependency Graph, where *MC* is the set of nodes (vertices) and E is the set of edges, $I \subseteq MC \times E$ is the incidence relation, for $\forall e \in E$ the $\{v \in MC : vIe\}$ set has two ordered elements, namely $a, b \in MC : vIa \wedge vIb \Leftrightarrow a \in dep_m(b)$, and $C \subseteq E \times \mathbb{N}$ is the coloring relation which assigns the same color to those edges which belong to the same full macro expansion. The E set contains multiple edges where each edge has a certain color, if several full expansions use the same edge. We use $dep_{m_i} \in dep_m$ to denote the subrelation colored with i: $\forall i \in \mathbb{N},\ b \in dep_{m_i}(a) \Leftrightarrow b \in dep_m(a) \wedge \exists e \in E : aIe \wedge bIe \wedge (e, i) \in C$.

It should be mentioned here that the *MDG* is an acyclic graph even when it contains subgraphs of the whole software system [7].

Generating macro slices can be performed on the *MDG*. A slicing criterion is a set $< p, x >$ where p is a program point and x is a variable at p. In the case of macro slicing the criterion is mapped to the *MDG*, and for the $< p, x >$ criterion there is a node $k \in MC$ in the dependency graph which represents the macro definition x at the program point p. The forward macro slice contains those program points which are reachable from k along colored edges in the graph.

Definition 3. Let $< p, x >$ be a slicing criterion where x is a definition at program point p and $k \in MC$ is the node corresponding to x. Let *Col* be the set of colors which are used on dependency edges starting from k:

$$Col = \{c \in \mathbb{N} | \exists e \in E, c \in C(e) \wedge (k, e) \in I\}.$$

The forward macro slice of the criterion is the set $S = \{y \in MC | y \in dep_{m_i}^{\ t}(k), i \in Col\}$, where $dep_{m_i}^{\ t}$ is the transitive closure of dep_{mi}.

Backward macro slices can be defined in a similar way, where the slice starts at a macro call and includes all definitions that are used during the full expansion of the macro.

A piece of source code and the associated dependency graph are given in Fig. 5. The dependency edge colors are represented as numbers. The forward macro slice based on the definition of Y as a criterion contains the definition of X and the second macro invocation X_2. A more detailed example is given in Section 6.

5.2. Connecting slices

The process of combining macro and language slices requires that a common set of nodes and edges be defined with the dependency relation as well. C/C++ language slices are usually computed on some kind of a Program Dependence Graph (PDG) [26], or more generally on a System Dependence Graph (*SDG*) introduced by Horwitz *et al* [9]. The *PDG* models interprocedural dependencies between procedures where each procedure is modelled with a *PDG*. In the preprocessor case, the *MDG* can be constructed in such a way that it contains dependencies from every compilation unit in a software; there is no need to define two kinds of graphs for the macros. In the following we shall consider a generalized *SDG*, on which a general C/C++ dependency relation is defined (called dep_{cc}).

The *MDG* can be used in combination with the *SDG* in the following way. Both of them have a well-defined structure, the only problematic point being the connection. The *MDG* is based on the original source code, while the *SDG* contains C/C++ language elements. In practice it is based on the preprocessed code (`.i` file). The toplevel macro invocation (call) serves as a connection point (see the motivating example in Section 3.1). From the point where the macro call is replaced with the replacement text, the source code is in C/C++ language form and consists of C/C++ program elements.

Unfortunately, there is no guarantee that the replacement text will be a C/C++ syntactical unit. Moreover, the *SDG* is composed of program elements, but contains various kinds of nodes like declaration, expression, return and so on. There is a many-to-many relation between macro replacement texts and *SDG* nodes. For instance the macro replacement may be a sequence of statements that is represented by more nodes in the *SDG*, and the macro may even be a constant which is only a part of an *SDG* node. An *SDG* node, which at least partially comes from a macro replacement, depends on the macro itself. Thus a dependency relation can be defined based on shared characters between the *SDG* node and the macro (replacement). Let $repl(a)$ be the replacement text after a full expansion of macro call a, where $repl(a)$ consists of characters with their position in the preprocessed file. (The *SDG* node b also contains characters with their position in the preprocessed file.)

Definition 4. Let $dep_{comb} \subseteq MDG \times SDG$, $a \in MDG$, $b \in SDG$, $b \in dep_{comb}(a)$ if and only if a is toplevel and $\exists x$ character: $x \in repl(a)$ and x is contained by b.

An *SDG* node depends on an *MDG* node if at least one of its characters comes from the replacement of the *MDG* node.

Using the definitions given in this section, the combined slice can be defined. The dep_{cc} C/C++ dependency relation, the dep_m macro dependency and the dep_{comb} combining dependency relations are already given. Next, let *DG* be the combined dependency graph and *dep* the combined dependency relation:

Definition 5. Let

$$DG = SDG \cup MDG$$

and

$$dep(x) \subseteq DG \times DG = \begin{cases} dep_m(x) \cup dep_{comb}(x), \text{ if } x \in MDG \\ {dep_{cc}}^{-1}(x), \text{ if } x \in SDG \end{cases}$$

Note that the *dep* relation uses the inverse of the dep_{cc} relation. In program slicing the direction of the dependency relations usually points in a backward direction. However, in the case of macro slicing the direction is the opposite of the macro call relation. To be consistent, for combined slicing the inverse of the C/C++ dependency should be used.

Definition 6. Let $< p, x >$ be a slicing criterion, where x is a variable at program point p. Let $k \in DG$ be the corresponding graph element for x. The combined forward slice of the criterion is the set of program points, which corresponds to the $\{l \in DG \mid l \in dep^t(k)\}$ set, where dep^t is the transitive closure of the *dep* relation.

Definition 7. Let $< p, x >$ be a slicing criterion, where x is a variable at program point p. Let $k \in DG$ be the corresponding graph element for x. The combined backward slice of the criterion is the set of program points, which corresponds to the $\{l \in DG \mid k \in dep^t(l)\}$ set, where dep^t is the transitive closure of the *dep* relation.

The forward direction is depicted in Fig. 6. The capital letters in the figure elements refer to their type and not their name. The slice starts at the slicing criterion, which is a macro definition (D). There is a set of dependent definitions (D), and there is a set of dependent toplevel macro invocations (T). (Note that many dependency edges among the elements of this set have been omitted here.) When toplevel invocations are replaced, the result of each invocation takes part in a set of C/C++ program elements (P). A regular language slicing algorithm computes the slice for each program element, hence the final combined slice contains every element in the figure.

The backward direction case is outlined in Fig. 7. Here once again the capital letters in the figure elements refer to their type and not their name. The slicing criterion is a C/C++ program element (P). The slice may contain *SDG* nodes which are (at least partially) the results of one or more macro invocations. The toplevel invocations which are present in the C/C++ slice can be found along the dependency edges. For all of these toplevel invocations, macro slice sets can be obtained using backward macro slicing. The final combined backward slice contains every element in the figure.

The combined graph and the combined slices of the sample source code from Section 3.1 can be seen in Fig. 8. Nodes belonging to forward and backward slices are denoted by a capital 'F' and 'B', respectively. The toplevel macro call

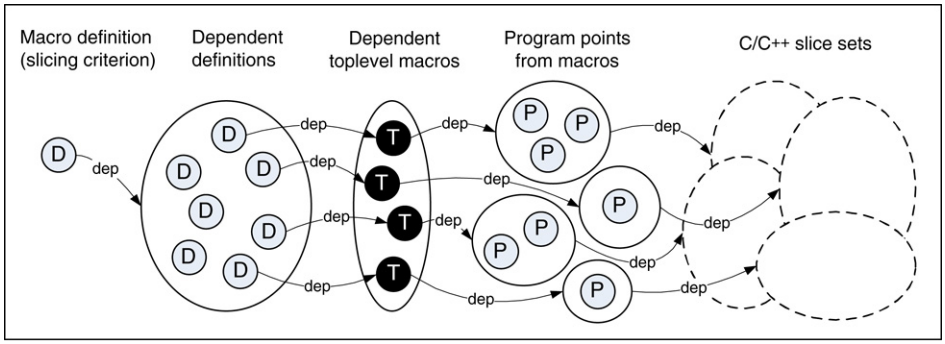

Fig. 6. The forward direction for combining the slices, with the dependency relation between macros and C/C++ program points.

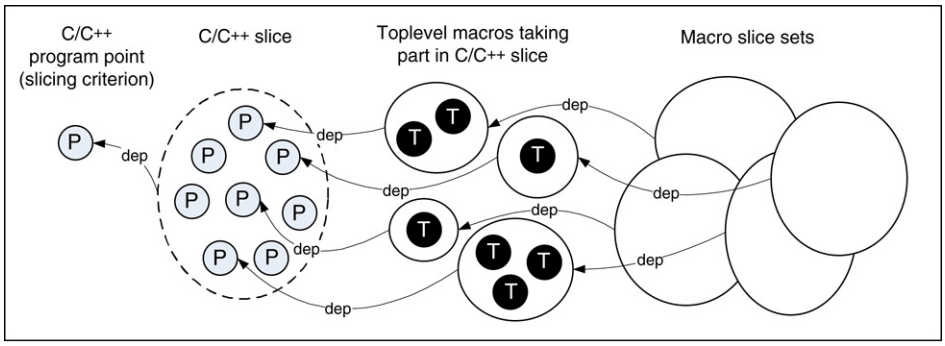

Fig. 7. The backward direction for combining the slices, with the dependency relation between macros and C/C++ program points.

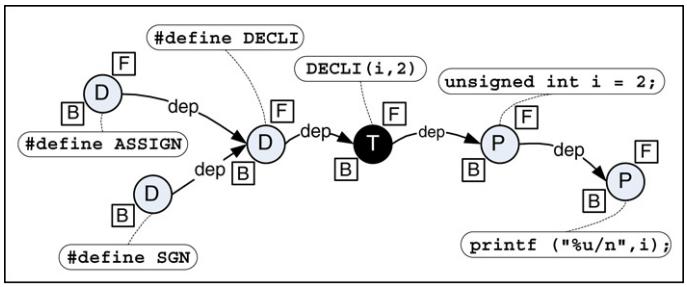

Fig. 8. Nodes and slices of the motivating example.

`DECLI(i,2)` is present in both of its possible forms: as a macro and as a program point. The forward slice contains every node except the definition of `SGN`, while the backward slice contains every node of the graph.

Note that the method does not make use of any special information concerning the *SDG* of the C/C++ slicing algorithm. Just the dependency relation and the character positions of the node texts are used. Hence, in theory the method can be used for static or dynamic slicing. Moreover, it does not matter whether data, control or some other dependency relation is used for slicing.

6. Tools and algorithms

The formal definitions of combined slices are given in the previous section. A combined slicer can be implemented in various ways. There are three tools that must be used to implement the method: a macro slicer, a C/C++ slicer, and a combiner tool which implements the connection between them. In this section we report our implementation. The algorithms given below follow the way how the tools are working. For the sake of extensive experiments, the slices are computed for each appropriate node in dependency graphs, so our tools and algorithms are global in this sense. To create an on demand version of the toolchain – which computes slices only for criteria given as input – minor changes are required (overviewed below).

6.1. Tool setup

In our toolchain the macro slicer is built on top of the Columbus framework [27,28]. The macro slicer tool analyses the project and afterwards creates a graph instance of the Columbus preprocessing schema [18]. The graph contains dependency

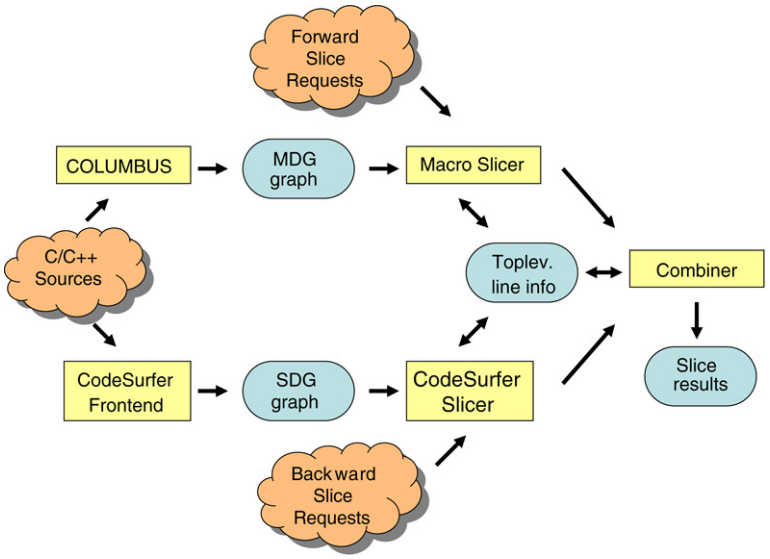

Fig. 9. Logical tool architecture — forward and backward slicing.

edges between preprocessor elements, therefore it can be used as an *MDG* on which macro slicing can be performed. For the C/C++ part we implemented a CodeSurfer plugin to get slicing information [6]. Similar to the previous case, CodeSurfer builds the *SDG* graph representation from a software project and determines language level dependencies (and other pieces of information as well). CodeSurfer gives access to the internal representation of the *SDG* and the dependency information via plugins. We used the C API, which just provides core functionality, but is suitable for slicing (the Scheme API provides full access).

The logical outline of the toolchain is depicted in Fig. 9. The toolchain consists of the core analyzers (Columbus and CodeSurfer), the macro slicer tool, the CodeSurfer slicer plugin and a small combiner tool which summarizes the results obtained (the combiner is implemented together with the macro slicer). The tools communicate with each other via a set of toplevel macros (given by their line information), which is the common point of the two slicers.

The process in both the forward and backward cases starts with the core analyzers. In the backward direction the C/C++ slices are continued with backward macro slices at points of macro calls. The CodeSurfer plugin produces the backward slice based on the criterion (which is a C/C++ program point). The slice is then scanned for vertices which are results of toplevel macro calls (matching). The slice is written into the output, and the set of toplevel macros present in the slice is given to the macro slicer tool. The macro slicer computes backward slice sets for each toplevel macro given as input, and by doing so extends the existing slice. Lastly the slices are summarized.

In the case of forward slicing the slicing criterion is a macro definition. The macro slicer produces the macro slice of the criterion, whose final result contains the set of toplevel macros, which is then given to the language slicer. In the next part the CodeSurfer plugin identifies positions in the source code where macro replacement was performed and the toplevel macros are matched with C/C++ vertices. The matching between toplevel macros and vertices is carried out based on line and column information (from the various types of vertices, just those which have a position in the source are used). Next, the language slicing algorithm is executed to produce slices for each vertex, which is then matched with toplevel macros. The results are summarized for each starting macro definition criterion (the C/C++ part of the final slice is the union of the C/C++ slices belonging to the toplevel macros).

In the following sections we will provide details about the implemented algorithms. The logical architecture shown in the previous section has been slightly altered: the combiner is implemented inside the macro slicer. Therefore two algorithms are used both in the backward and forward case: one for the CodeSurfer plugin and one for the macro slicer and combiner. The CodeSurfer plugin is first run in both cases followed by the macro slicer and combiner. The following notation is used in the algorithm descriptions: the *Cs* and *M* prefixes refer to CodeSurfer (C/C++) and Macro artifacts, respectively; *vertex* means a node in the *SDG*, while *toplev* means a toplevel macro invocation.

6.2. Backward algorithm

Our combined backward slicing algorithm is given in Fig. 10. As mentioned before, the backward direction means that the C/C++ slices are continued with backward macro slices at points of toplevel macro calls. The plugin gets the vertices from each procedure and then computes the backward C/C++ slice on the project *SDG* (the function *GetProcedureVertices*(*SDG*) returns vertices contained in procedures which have source file positions). Each such slice is scanned one vertex at a time, and the set of matching toplevel macros is found. The *Match*(*y*, *AllToplevs*) function returns the matching toplevel macro set for a vertex (*AllToplevs* denotes the set of all toplevel macros, for matching see Section 6.4). The toplevel macros are

CodeSurfer plugin - Backward slice
input: SDG : SDG of the analyzed project
output: $outS$: set of $< v, CsBwSlice_v, T_v >$ triplets where:
v : vertex $\in SDG$
$CsBwSlice_v$: backward C/C++ slice of v
T_v : set of toplevel macros in $CsBwSlice_v$
begin
1 $outS = \emptyset$
2 **foreach** $v \in GetProcedureVertices(SDG)$
3 $T_v = \emptyset$
4 $CsBwSlice_v$ = compute backward C/C++ slice for v on SDG
5 **foreach** $y \in CSBwSlice_v$
6 $T_v = T_v \cup Match(y, AllToplevs)$
7 $outS = outS \cup < v, CsBwSlice_v, T_v >$
end

MacroSlicer & Combiner - Backward slice
input: MDG : MDG of the analyzed project
inS : set of $< v, CsBackSlice_v, T_v >$ triplets
output: S : set of $< v, S_v >$: pairs - combined slice set
for each request (vertex)
begin
1 $S = \emptyset$
2 **foreach** $< v, CsBackSlice_v, T_v > \in inS$
3 $S_v = CsBackSlice_v$
4 **foreach** $x \in T_v$
5 $MBwSlice_x$ = compute backward macro slice for x on MDG
6 $S_v = S_v \cup MBwSlice_x$
7 $S = S \cup < v, S_v >$
end

Fig. 10. Computing combined backward slice.

combined for each such C/C++ slice. The triplet with the original vertex, the associated C/C++ slice and the set of toplevel macros are computed for each criterion and the result is passed to the macro slicer and combiner.

In the second step the macro slicer and combiner produces the final slices for each vertex passed as input. First, the C/C++ slice is part of the final slice. Second, the set of included toplevel macros is used to compute additional backward macro slices. These macro slices are then placed in the final slice set. The result is the combined backward slice.

In the backward direction the toolchain may work in an on-demand way; in this case the plugin in line 2 of the algorithm iterates through the vertex set passed as an argument.

6.3. Forward algorithm

In the forward direction the slicing criterion is a macro definition. The forward macro slices are combined with C/C++ slices via toplevel macros matched with *SDG* vertices. In this direction the toolchain acts as a global slicer. The CodeSurfer plugin prepares toplevel macros and the associated C/C++ slices for the whole program. The prepared data is passed to the macro slicer and combiner, which computes macro slices and creates the final sets.

Fig. 11 lists the combined forward slicing algorithm employed. The CodeSurfer plugin iterates through all vertices inside procedures and tries to find matching toplevel macros. In the case of a successful match, the forward C/C++ slice of the current vertex is computed, and the set of matched toplevels is paired with the C/C++ slice. The output of the plugin is the set of toplevels paired with the forward slices starting from the matched vertices. The macro slicer and combiner iterates through all macro definitions in the *MDG* (with the help of the *GetDefinitions*() function). For each definition the forward macro slice is computed, which will be part of the final combined slice. The macro slice contains (usually several) toplevel macros (provided by the *GetToplevs*() function). For each included toplevel macro in the macro slice (*GetToplevs*() function), the set of C/C++ slices is got from the input (*GetCsFwSlice*() function) and then added to the combined slice. The final result is the set of combined slices paired with the associated definition.

Creating an on-demand slicer requires that the macro slicer and the combiner be separated and the tools be called in the following order: macro slicer (with input criteria), plugin, combiner.

6.4. Details on matching and graph coloring

There are many factors which make the matching of macros and vertices based on file position a challenging task. The behaviour of the tools had to be adjusted in many areas including the physical and logical lines (e.g. for the `#line` directive

CodeSurfer plugin - Forward slice
input: SDG : SDG of the analyzed project
output: $outS$: set of $< T , CsFwSlice_T >$ pairs where:
T : set of toplevel macros
$CsFwSlice_T$: forward C/C++ slice connected to T
begin
1 $outS = \emptyset$
2 **foreach** $v \in GetProcedureVertices(SDG)$
3 **if** $Match(v, AllToplevs) \neq \emptyset$
4 $CsFwSlice_v$ = compute forward C/C++ slice for v on SDG
5 $outS = outS \cup < Match(v, AllToplevs) , CsFwSlice_v >$
end

MacroSlicer & Combiner - Forward slice
input: MDG : MDG of the analyzed project
inS : set of $< T , CsFwSlice_T >$ pairs
output: S : set of $< d , S_d >$: pairs - combined slice set
for each request (macro definition)
begin
1 $S = \emptyset$
2 **foreach** $d \in GetDefinitions(MDG)$
3 $MFwSlice_d$ = compute forward macro slice for d on MDG
4 $S_d = MFwSlice_d$
5 **foreach** $t \in GetToplevs(MFwSlice_d)$
6 $S_d = S_d \cup GetCsFwSlice(inS, t)$
7 $S = S \cup < d , S_d >$
end

Fig. 11. Combined forward slice algorithm.

CodeSurfer preserves the original line information), handling macros in conditional directives, and handling macros defined in the command line. The plugin iterates through vertices belonging to procedures, which means that some vertices are omitted such as forward declarations or globals). Another important factor is the handling of standard libraries. The *SDG* contains additional vertices from standard libraries, and some vertices used in its internal representation. Accordingly, the macro slicing tool is adjusted to match macros from standard libraries, but not to report errors for omitted ones.

The matching process is based on comparing source position intervals. The result of the *Match* (*vertex*: y , *set* $<$ *toplev* $>$: T) function is the subset of T. The *repl*(a) function gives the replacement text after a full expansion of macro call a, where *repl*(a) consists of characters with their position in the preprocessed file. If the vertex y contains characters from *repl*(m) (i.e. the expansion of the toplevel macro $m \in T$), then the matching set contains m. In other words, the matching algorithm checks the file position of the vertex and the replacement of macros, and if there are overlapping intervals then the matching is successful.

A schematic view of the matching process is shown in Fig. 12. The toplevel macro (T) is expanded using two definitions (D_1, D_2). The final replacement is denoted by *repl*(T) in the figure. The result is included in the C/C++ analysis, and the *SDG* vertices are defined based on the preprocessed source including *repl*(T). Vertices are denoted by horizontal lines as they may overlap the same source position. The figure contains two successful matches, namely (T) is matched with both (P_1, P_2). Note that the replacement text is included in matching in full length. Lastly, the combined forward slice requested on D_2 consists of $\{(D_2, D_1), T, (P_1, P_2)\}$.

The matching algorithm can be refined with more accurate check on positions. If we track the origin of the pieces contained by the replacement text, then the slice set may be smaller. In this case P_2 is matched with pieces from both D_1 and D_2, but P_1 matches only pieces from D_1. Therefore using accurate tracking the combined forward slice on D_2 does not contain P_1, it consists of $\{(D_2, D_1), T, (P_2)\}$. This kind of slicing produces smaller, more accurate slices. However the result is not necessarily better (it is not obvious that P_1 is not related to D_2). Another question arises about the interpretation: should D_1 be contained by forward slice of D_2? The toolchain used in our experiments used the first type of matching without tracking macro pieces.

The graph coloring method is introduced briefly in Section 5. An illustrative example is given in Fig. 13. The program code is given on the left-hand side of the figure, which is followed by the basic and the colored version of the graph (colors are represented by solid, dotted and dashed lines). Coloring reflects the full macro expansion of toplevel macros. For example macro call A_1 uses the definition of A (solid lines). During the further expansion macro B is also expanded, but C is not defined at that source position. Using the basic dependency graph for macro slicing would result inaccurate (larger) slices. Computing forward macro slice on the definition of E using the basic graph would result in E, C, D, D_1, A, A_1, A_2. Using the dashed edges in the colored graph a much better slice can be computed, namely E, C, D, D_1. Coloring helps in a similar way in the case of backward macro slicing. The backward macro slice computed on A_1 using the basic graph includes unnecessary nodes

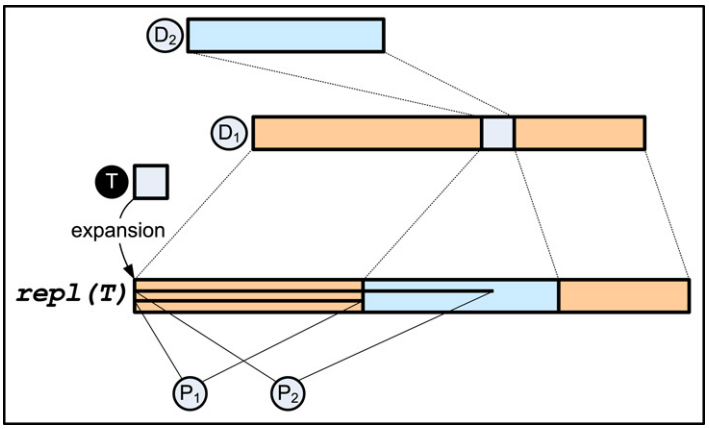

Fig. 12. Matching based on common characters in an expansion.

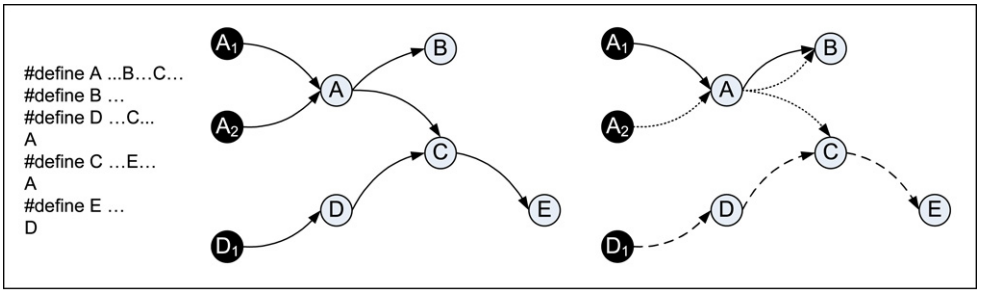

Fig. 13. Edge coloring example.

(C, E). Although the presented example is artificial, the analyzed projects contain several complex preprocessor constructs, which confirms the necessity of graph coloring.

7. Measurements

7.1. Subject programs

Experiments are performed on 28 open source projects, starting from small programs to medium size ones to about 20k lines of code. Many of the programs are selected based on remarkable empirical studies on slicing [19] and preprocessor usage [29]. We found the total of 240k nonempty lines of code enough to prove the usability of the method. Table 1 contains the list of projects used in our measurements, and their basic data. Sizes given in nonempty lines of code as CodeSurfer calculates its LCode metric (note that this metric is significantly smaller than the usual LOC metric, when usually comments and empty lines are counted). Build time of dependency graphs is given in seconds, as the *time* unix tool reports the user time of the process. The building time includes the time needed to build the project, not only the graph building phase. The number of nodes in the graphs can be found as a measure for the graph size. Not surprisingly, the *MDG* is smaller than the *SDG*, which is almost 60 times larger on average. The time required for slicing operation is given in the tables, backward and forward slicing is done during the same run. The memory consumption was below 350M for the CodeSurfer plugin and below 2.5G for the macros slicer and combiner tool (without any special effort spent on decreasing memory consumption).

7.2. Slices in detail

In our experiments the measure for the slice size was the number of source code lines which contain vertices from the slice, since this seems to be the best common denominator for the different slicing tools. Other researchers also used this approach [19].

Because of the difficulties in matching, which were mentioned in the previous section, there were slices in both directions which the tools failed to match. The failure rate was generally about 8% in the forward case, and under 1% in the backward case, which we found acceptable for reporting measured data. The data given in this section just contains the perfectly matched slices.

The left-hand side of Table 2 contains the number of combined forward slices and their average sizes. We computed all possible forward slices, meaning that we started from each macro definition, and measured the sizes of the individual macro

Table 1
Subject programs.

Program name	Size (LCode)	MDG build time (s)	MDG size (nodes)	SDG build time (s)	SDG size (nodes)	Macro sl. time (s)	C/C++ sl. time (s)
replace	512	0.28	136	1.18	3 205	0.26	7.85
copia	1 085	0.45	7	6.13	94 390	0.12	208.65
time	1 119	1.88	162	4.15	5 633	0.26	3.73
which	1 246	1.87	146	5.41	7 449	0.48	29.44
compress	1 335	0.84	108	2.18	4 408	0.16	8.29
wdiff	1 364	2.12	217	4.57	7 640	0.53	10.77
ed	2 637	3.80	117	9.98	39 412	0.73	716.82
barcode	2 807	6.34	381	13.76	27 970	3.1	427.62
tile	3 549	1.93	1 881	27.69	51 095	19.72	146.43
acct	4 008	9.37	899	12.50	24 619	5.0	116.98
li	4 793	10.71	1 826	3006.31	943 340	79.9	56 238.38
EPWIC	5 249	12.10	852	14.68	27 099	12.23	443.48
lightning	5 563	20.8	1 750	69.42	56 778	6 954.21	572.75
gzip	5 997	9.88	1 725	17.88	37 525	34.16	1 315.92
userv	6 016	5.47	1 244	24.72	105 902	23.30	3 281.28
indent	7 582	4.55	857	12.22	42 102	17.98	1 100.14
bc	9 472	9.6	1 554	24.90	59 503	31.17	2 080.13
diffutils	10 124	18.91	1 971	29.35	53 928	31.54	1 261.76
gnuchess	11 045	13.87	2 511	29.12	70 782	143.8	4 391.19
ctags	11 670	12.96	1 480	55.31	209 357	106.61	12 611.60
sed	13 339	9.37	2 527	26.28	89 788	204.76	9 374.67
nano	13 698	14.96	3 964	38.11	177 879	591.88	23 445.10
ijpeg	15 253	25.82	4 283	39.75	77 531	212.62	6 948.48
flex	17 533	22.56	3 188	112.12	126 757	259.55	9 912.45
bison	20 673	35.74	4 387	88.64	138 972	98.92	16 099.25
wget	21 104	27.88	4 146	95.28	269 209	993.85	60 294.88
espresso	21 780	3.86	0	52.79	151 802	0.18	9 642.20
go	22 118	5.40	5 296	22.18	110 236	499.19	22 550.61
Total	242 671	293.32	47 615	3846.61	3 014 311	10 326.21	243 240.85

and language slices along with the combined slices. The numbers listed are the average slice size values. We treat the set of toplevel macros in a special way so we count the toplevel macros into both the macro slice and the associated vertices into the C/C++ slice as they belong to both kinds of slices.

There are two items of special interest in the list. The program *espresso* is interesting because it does not contain any macro definitions. The program *lightning* is exactly the opposite: it is the only one that has larger macro slices than C/C++ slices. Examining the code confirms that some C source files of this program are full of macro definitions and calls.

Backward slices may not necessarily contain macro calls. Although the average number of macro calls is not so high, most of the backward slices contain macro calls (above 75%). The number of combined slices (which necessarily contain macros) and their average sizes are given in Table 2 (on the right-hand side), where we used the same approach for measurement as we did with the forward slices. It can be seen that backward macro slices are generally bigger than forward slices, which can be explained by the fact that language slices usually contain many more code lines and hence more potential starting points for macro slices exist (we used both data and control dependencies for slicing C code). Another reason might be that in the backward case we produce slices for each vertex, so more of the large slices are counted, while in the forward case we selected just a few vertices (according to the macro calls). This way, the average may be higher in the backward case.

Studying the ratio of macro slice sizes relative to the C language slice sizes it can be seen that the individual macro slices are relatively small, but this may be due to the size difference of the SDG and the MDG graphs. For a given slicing criterion the smaller the slice the better, naturally without ignoring any dependency. Macro slices are more accurate in this sense, while still having relatively small additional percentage value.

There is a wide range of open source software which has been analyzed by Ernst et al [29]. They report the preprocessor directive usage in open source software and find that preprocessor directives make up about 8.4% of the program code on average. It is worth mentioning that in both directions the extra code lines coming from macro slices are relatively small compared to the language slices, so their true worth is debatable here. However, we think that in many cases these additions may be crucial from a program comprehension point of view. The real world example in Section 3.2 provides an example where the macro slice part is small but useful. This is not a rare occurrence. The example is taken from the *flex* program where macro usage is close to the average according to the empirical study mentioned above. In this respect, traditional C/C++ language slices without macro slices can be treated as *unsafe*, overlooking important information.

8. Conclusions and future work

The work presented was motivated by the observation that virtually all available program slicing tools for the C/C++ language lack the proper and complete handling of preprocessor constructs. From a program comprehension point of view,

Table 2
Summary of forward and backward slices (slice sizes are given in source code lines).

	Forward slices				Backward slices			
	Number of slices	Macro sl. size (avg)	C lang sl. size (avg)	Comb. sl. size (avg)	Number of slices	C lang sl. size (avg)	Macro sl. size (avg)	Comb. sl. size (avg)
replace	23	7.1	328.7	335.9	647	205.2	86.6	291.8
copia	2	3.0	1132.5	1135.5	3 044	924.2	6.0	930.2
time	31	8.9	287.6	296.5	598	115.4	19.0	134.4
which	22	6.3	544.8	551.1	1 288	396.1	53.6	449.8
compress	26	4.7	277.9	282.6	601	323.7	64.5	388.2
wdiff	25	7.0	300.6	307.6	989	170.2	45.4	215.6
ed	27	3.3	1459.2	1462.5	3 849	1543.8	37.5	1581.3
barcode	44	6.7	1665.8	1672.5	4 143	1569.3	153.1	1722.3
tile	145	23.1	2468.0	2491.0	2 469	358.9	193.3	552.2
acct	76	14.2	761.9	776.1	3 896	492.1	93.8	585.9
li	111	29.7	3966.6	3996.3	7 695	4025.3	1392.2	5417.5
EPWIC	122	7.1	1102.5	1109.6	6 434	897.0	239.7	1136.7
lightning	341	983.8	167.5	1151.3	808	101.0	73.0	174.0
gzip	259	11.0	3274.0	3285.0	4 701	2884.3	979.7	3864.1
userv	202	12.2	2732.3	2744.5	8 002	2163.5	482.7	2646.2
indent	53	16.8	4518.0	4534.8	5 781	3196.4	427.3	3623.7
bc	153	9.3	3832.7	3842.1	8 548	3108.0	612.7	3720.6
diffutils	242	13.7	2997.5	3011.2	9 614	1830.4	397.4	2227.8
gnuchess	242	14.8	7086.4	7101.2	10 919	5137.0	1838.9	6975.8
ctags	111	17.0	8337.1	8354.1	12 775	7496.0	976.1	8472.0
sed	256	102.6	8812.6	8915.2	12 295	7367.3	1571.9	8939.2
nano	389	18.3	12590.1	12608.4	14754	12750.5	3354.6	16105.1
ijpeg	322	19.1	5861.3	5880.4	13 479	5661.1	1859.5	7520.6
flex	334	16.3	9036.2	9052.5	12 530	7250.8	1977.1	9227.9
bison	248	28.0	4458.6	4486.6	3 873	5763.4	1792.0	7555.4
wget	340	27.2	16045.4	16072.6	21 471	14345.1	2838.9	17184.0
espresso	0	nan	nan	nan	0	nan	nan	nan
go	382	38.2	10817.2	10855.4	23 840	10304.6	3628.1	13932.7
Total/avg	4528	∼ 96.62	∼ 6525.98	∼ 6622.60	199 043	∼ 6647.69	∼ 1706.99	∼ 8354.68

the existing methods often appears inadequate. For instance, the impact of changing a macro definition cannot be accurately followed throughout the program's preprocessor and non-preprocessor-related parts. Existing tools either compute the slices based on dependencies in the language constructs or provide rich features to model macro usage, but not both. This could have a detrimental impact on various fields related to program comprehension and maintenance in general. For example, in change impact analysis, a failure to identify a dependency of a change could have the effect of inaccurately predicting the cost of changes and of performing incomplete change propagation, which in turn would result in increased risk of regression [30].

With this work we fill this gap and propose a *combined* approach for computing slices in C/C++ programs. We justify out the approach by providing a realistic sample program comprehension problem and other possible applications of the method. Existing tools were employed in an experimental tool setup with which a number of program slices were computed. We counted the program points returned by the combined approach and compared it to slices without the preprocessor components.

The first results measured on open source projects look promising, and clearly demonstrate the benefits of using our approach. However, the method needs to be refined and larger case studies should be performed. We plan to qualitatively evaluate the approach in greater depth to look for usage scenarios where it would be most beneficial. We also plan to conduct more experiments with much bigger systems, like the Mozilla source code. But even at this stage of the research we definitely recommend that similar combined strategies for slice calculation in existing tools like CodeSurfer be integrated. In it, one may be able to use and extend the existing internal representation for this purpose. Finally, extending CodeSurfer's GUI capabilities with combined slice features is an interesting idea. We intend to pursue these tasks in the near future as well.

Acknowledgements

We would like to thank Judit Jász for her help in programming CodeSurfer. This research was supported in part by the Hungarian national grants RET-07/2005, OTKA K-73688 and TECH_08-A2/2-2008-0089.

References

[1] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.
[2] M. Weiser, Program slicing, IEEE Transactions on Software Engineering SE-10 (4) (1984) 352–357.

[3] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, ACM SIGSOFT Software Engineering Notes 30 (2) (2005) 1–36.
[4] A. Garrido, Program refactoring in the presence of preprocessor directives, Ph.D. Thesis, UIUC, 2005.
[5] M. Vittek, P. Borovanský, P.-E. Moreau, A collection of C, C++ and Java code understanding and refactoring plugins, in: ICSM (Industrial and Tool Volume), 2005, pp. 61–64.
[6] GrammaTech CodeSurfer Homepage, Homepage of GrammaTech's CodeSurfer, 2008. URL http://www.grammatech.com/products/codesurfer.
[7] L. Vidács, A. Beszédes, R. Ferenc, Macro impact analysis using macro slicing, in: Proceedings of the Second International Conference on Software and Data Technologies, ICSOFT'07, 2007, pp. 230–235.
[8] L. Vidács, J. Jász, Á. Beszédes, T. Gyimóthy, Combining preprocessor slicing with C/C++ language slicing, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, ICPC'08, 2008, pp. 163–171.
[9] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–61.
[10] H. Agrawal, J.R. Horgan, Dynamic program slicing, in: Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, SIGPLAN Notices (6) (1990) 246–256.
[11] S.A. Bohner, R.S. Arnold (Eds.), Software Change Impact Analysis, IEEE Computer Society Press, ISBN: 0818673842, 1996.
[12] K.B. Gallagher, J.R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (8) (1991) 751–761.
[13] A. Cimitile, A. de Lucia, M. Munro, A specification driven slicing process for identifying reusable functions, Journal of Software Maintenance: Research and Practice 8 (3) (1996) 145–178.
[14] J. Zhao, A slicing-based approach to extracting reusable software architectures, in: Proceedings of the 4th European Conference on Software Maintenance and Reengineering, CSMR'00, 2000, pp. 215–223.
[15] H. Agrawal, R.A. DeMillo, E.H. Spafford, Debugging with dynamic slicing and backtracking, Software — Practice and Experience (SPE) 23 (6) (1993) 589–616.
[16] D.W. Binkley, The application of program slicing to regression testing, Information and Software Technology 40 (11–12) (1998) 583–594.
[17] G. Rothermel, M.J. Harrold, Selecting tests and identifying test coverage requirements for modified software, in: Proceedings of ISSTA'94, 1994, pp. 169–183.
[18] L. Vidács, A. Beszédes, R. Ferenc, Columbus Schema for C/C++ Preprocessing, in: Proceedings of CSMR 2004, IEEE Computer Society, 2004, pp. 75–84.
[19] D. Binkley, M. Harman, A large-scale empirical study of forward and backward static slice size and context sensitivity, in: Proceedings of the International Conference on Software Maintenance, ICSM'03, IEEE Computer Society, 2003, pp. 44–53.
[20] Unravel Homepage, Homepage of the Unravel project, 2008. URL http://itl.nist.gov/div897/sqg/unravel/unravel.html.
[21] M. Mock, D.C. Atkinson, C. Chambers, S.J. Eggers, Program slicing with dynamic points-to sets, IEEE Transactions on Software Engineering 31 (8) (2005) 657–678.
[22] D. Binkley, M. Harman, Locating dependence clusters and dependence pollution, in: Proceedings of the 21st International Conference on Software Maintenance, ICSM'05, IEEE Computer Society, 2005, pp. 177–186.
[23] P.E. Livadas, D.T. Small, Understanding code containing preprocessor constructs, in: Proceedings of IWPC 1994, IEEE Computer Society, 1994, pp. 89–97.
[24] B. Kullbach, V. Riediger, Folding: An approach to enable program understanding of preprocessed languages, in: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001), IEEE Computer Society, Los Alamitos, 2001, pp. 3–12.
[25] Understand for C++ Homepage, Homepage for Understand C++, 2007. URL http://www/scitools.com.
[26] K.J. Ottenstein, L.M. Ottenstein, The program dependence graph in a software development environment, in: Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE), Pittsburgh, Pennsylvania, SIGPLAN Notices (19(5)) (1984) 177–184.
[27] R. Ferenc, Á. Beszédes, M. Tarkiainen, T. Gyimóthy, Columbus – Reverse Engineering Tool and Schema for C++, in: Proceedings of the 18th International Conference on Software Maintenance, ICSM 2002, IEEE Computer Society, 2002, pp. 172–181.
[28] FrontEndART Software Ltd., FrontEndART Software Ltd., 2008. URL http://www.frontendart.com.
[29] M.D. Ernst, G.J. Badros, D. Notkin, An empirical analysis of C preprocessor use, IEEE Transactions on Software Engineering 28 (12) (2002) 1146–1170.
[30] V. Rajlich, A model for change propagation based on graph rewriting, in: Proceedings of ICSM 1997, 1997, pp. 84–91.